Dataflow-based Pruning for Speeding up Superoptimization

MANASIJ MUKHERJEE, University of Utah
PRANAV KANT, University of Utah
ZHENGYANG LIU, University of Utah
JOHN REGEHR, University of Utah

Superoptimization is a compilation strategy that uses search to improve code quality, rather than relying on a canned sequence of transformations, as traditional optimizing compilers do. This search can be seen as a program synthesis problem: from unoptimized code serving as a specification, the synthesis procedure attempts to create a more efficient implementation. An important family of synthesis algorithms works by enumerating candidates and then successively checking if each refines the specification, using an SMT solver. The contribution of this paper is a pruning technique which reduces the enumerative search space using fast dataflow-based techniques to discard synthesis candidates that contain symbolic constants and uninstantiated instructions. We demonstrate the effectiveness of this technique by improving the runtime of an enumerative synthesis procedure in the Souper superoptimizer for the LLVM intermediate representation. The techniques presented in this paper eliminate 65% of the solver calls made by Souper, making it 2.32x faster (14.54 hours vs 33.76 hours baseline, on a large multicore) at solving all 269,113 synthesis problems that Souper encounters when optimizing the C and C++ programs from SPEC CPU 2017.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; *Translator writing systems and compiler generators*; *Automated static analysis.*

Additional Key Words and Phrases: program synthesis, abstract interpretation, superoptimization, pruning

ACM Reference Format:
Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-based Pruning for Speeding up Superoptimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 177 (November 2020), 24 pages. https://doi.org/10.1145/3428245

1 INTRODUCTION

Synthesis is hard to scale up because the search space grows as a large function of the size of the synthesized code. The essence of scalable synthesis, then, is cutting down the search space using techniques such as divide-and-conquer [Alur et al. 2017], sketching [Bornholt et al. 2016; Solar-Lezama 2008], syntax-guided synthesis [Alur et al. 2013], or a type-guided approach [Guo et al. 2019]. This paper attacks scalability problems in enumerative synthesis where *candidates* are created—typically in order of decreasing desirability—and successively rejected using an SMT solver until a candidate is found that can be verified to refine the *specification.* The context for our work is a *synthesizing superoptimizer* for code in the LLVM intermediate representation (IR): this tool's goal is to automatically discover optimizations similar to those implemented by hand in LLVM's peephole optimization passes.

Authors' addresses: Manasij Mukherjee, University of Utah, manasij@cs.utah.edu; Pranav Kant, University of Utah, pranavk@cs.utah.edu; Zhengyang Liu, University of Utah, liuz@cs.utah.edu; John Regehr, University of Utah, regehr@cs.utah.edu.

2475-1421/2020/11-ART177
https://doi.org/10.1145/3428245

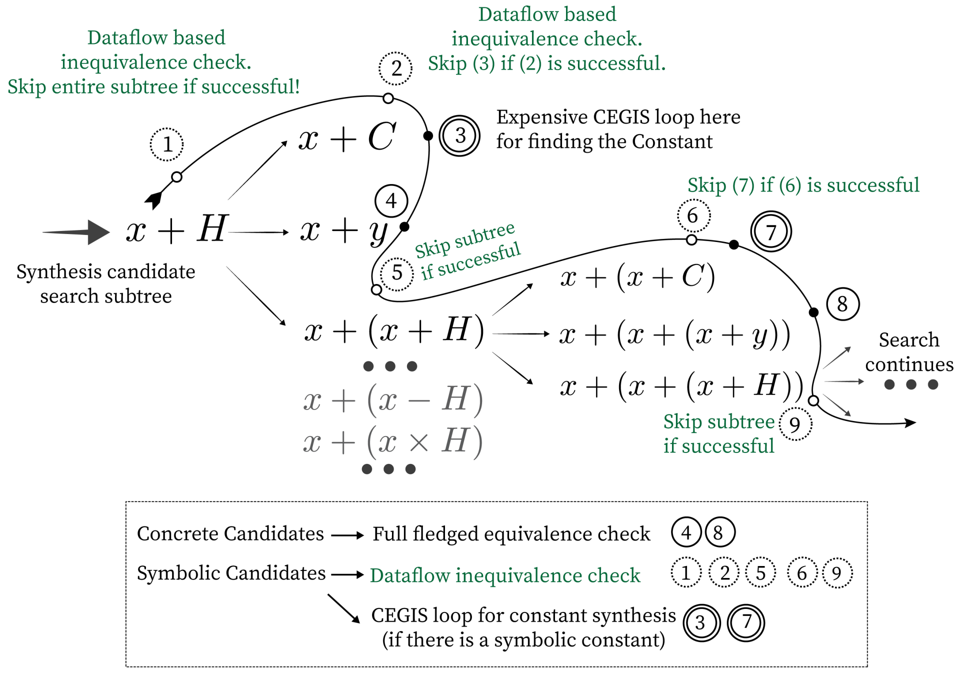
Dataflow based inequivalence check. Skip entire subtree if successful!
Dataflow based inequivalence check. Skip (3) if (2) is successful.
Expensive CEGIS loop here for finding the Constant
Skip (7) if (6) is successful
Skip subtree if successful
Synthesis candidate search subtree
$x + H$
$x + C$
$x + y$
$x + (x + H)$
$x + (x - H)$
$x + (x \times H)$
$x + (x + C)$
$x + (x + (x + y))$
$x + (x + (x + H))$
Search continues
Skip subtree if successful
Concrete Candidates → Full fledged equivalence check 4 8
Symbolic Candidates → Dataflow inequivalence check 1 2 5 6 9
CEGIS loop for constant synthesis (if there is a symbolic constant) 3 7

Fig. 1. Overview of enumerative synthesis with partially symbolic candidates and pruning. Successful pruning checks enable the search procedure to skip entire subtrees or expensive constant synthesis procedures. H stands for a *hole* and C stands for a symbolic constant.

Enumerative synthesis cannot scale without being able to rapidly reject incorrect candidates. The obvious way to do this is to interpret each candidate on several different inputs, rejecting it as soon as its behavior deviates from the behavior of the specification. Empirically, a large fraction of candidates can be rejected this way using only a few choices of inputs, greatly reducing the number of times an SMT solver must be invoked. Unfortunately, synthesis by enumerating fully concrete candidates cannot succeed when synthesized code may contain large constant values because, for example, up to 2^{64} alternatives must be considered in order to synthesize a single 64-bit constant.

The enumerative synthesizer that our work builds on is part of the open-source superoptimizer Souper [Sasnauskas et al. 2017], which extracts DAGs of LLVM instructions into its own IR. These then serve as the specifications that Souper attempts to improve upon using synthesis. Souper enumerates *partially symbolic* candidates such as $x + C$ where x is an input variable and C is a symbolic constant. It then uses a CEGIS [Gulwani et al. 2011] loop to find a value for C which works for all values of x. Pruning a candidate like $x + C$ is challenging because to do so, we must prove that it cannot refine the specification for any choice of C. Moreover, Souper's enumerator is partially lazy; it produces candidates such as $x+H$, where H is a *hole* representing an arbitrary DAG of not-yet-enumerated instructions. Pruning a candidate containing a hole is highly worthwhile, because this eliminates an entire subtree of the search space, but it is difficult because holes are "powerful"—they can perform a wide variety of computations. Enumeration and the role of pruning are illustrated in Figure 1.

The main contribution of this paper is a framework for pruning synthesis candidates using forward and backward dataflow analyses, to compute properties of the specification and candidate

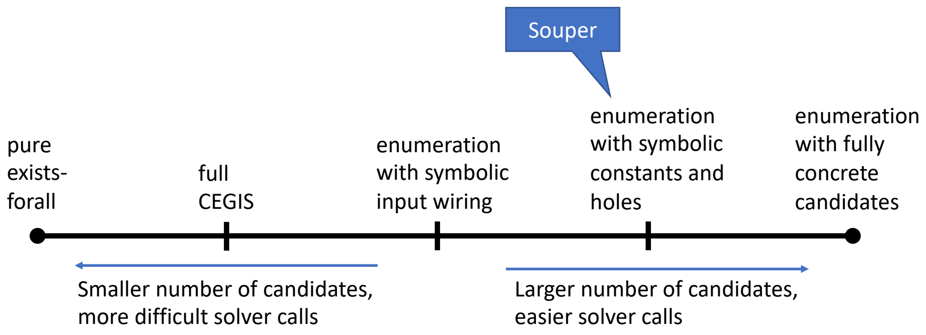
Souper
pure exists-forall
full CEGIS
enumeration with symbolic input wiring
enumeration with symbolic constants and holes
enumeration with fully concrete candidates
Smaller number of candidates, more difficult solver calls
Larger number of candidates, easier solver calls

Fig. 2. One way to look at different synthesis strategies is to ask how much of the search is explicit, and how much is pushed into the SMT solver

with the goal of establishing a *conflict* between them. A conflict corresponds to a proof that no concrete instantiation of the partially symbolic candidate can refine the specification. We have developed four novel pruning methods that fit within this framework, and we implemented a fifth method that appeared in previous work. We show that they are, individually and together, effective at reducing the number of times an SMT solver must be invoked and also at reducing total synthesis time. Enabling these pruning techniques reduces the time for Souper to run synthesis on all candidates from SPEC CPU 2017 (on a large multicore machine) from 33.8 hours to 14.5 hours, a speedup of 2.3x. At the same time, pruning increases the number of synthesis successes (where a candidate that is cheaper than the specification is found) by 8%, because making synthesis faster sometimes makes the difference between completing before or after a timeout. We believe that our dataflow-based pruning techniques should be broadly applicable to enumerative synthesis for bitvector problems.

2 BACKGROUND

2.1 Souper

Peephole optimizations are local rewrites that improve efficiency and code size; a canonical example is exploiting one of De Morgan's Laws to rewrite $(\neg P) \wedge (\neg Q)$ as $\neg(P \vee Q)$, saving one operation. The LLVM compiler contains a significant amount of implementation complexity related to peephole optimizations. These can be found in several optimization passes, but the most important one is the "instruction combiner," implemented as 36,000 lines of C++.

The Souper superoptimizer [Sasnauskas et al. 2017] represents an effort to automatically derive peephole optimizations using synthesis, with the long-term goal of replacing artifacts such as the instruction combiner. One potential advantage of this approach is a reduction in compiler defects: optimizations discovered by Souper are formally verified, whereas the instruction combiner was the buggiest element of LLVM according to Csmith [Yang et al. 2011]. Another advantage is rapid adaptation to novel code patterns emitted by new LLVM frontends and new application domains. In contrast, peepholes specified by hand take considerable time and engineering effort to adapt to new situations.

Every integer SSA value that Souper finds in a function represented in the LLVM intermediate representation (IR) is extracted into Souper IR using backwards slicing along dataflow edges. There is no requirement that extracted instructions are adjacent, or even that they come from the same basic block. This backwards slicing has several termination conditions: it stops when arriving at a function return value or function entry point (i.e., Souper is not interprocedural), a load from memory, or a loop back edge. When slicing passes a conditional branch, Souper extracts a *path condition*—a predicate indicating what information was learned from the branch condition. In other words, if some LLVM code branches on the condition "x is non-negative," then when Souper optimizes an SSA value in the conditional code, it can take advantage of this possibly-helpful information. A path condition effectively serves as part of the precondition for an optimization derived by Souper.

For every specification extracted from LLVM IR, Souper attempts to synthesize a more efficient computation that refines the original code. When one is found, Souper rewrites the LLVM IR to use the new way to compute the value being optimized.

2.2 A spectrum of search strategies for synthesis

Synthesis involves navigating very large search spaces. A practical problem for any synthesis effort is how much of the search to perform explicitly, and how much to push into the SMT solver. This is a complex issue and, as far as we know, no real consensus about how to best solve it exists within the research community. The simplest option is to exhaustively enumerate concrete candidates; a candidate is fully concrete when it needs no further elaboration to be executable. This corresponds to the right-most design point shown in Figure 2, and results in the largest number of candidates being enumerated. An advantage of concrete candidates is that they are often easy to prune without consulting a solver; Bansal and Aiken's superoptimizer [Bansal and Aiken 2006] and part of the LENS [Phothilimthana et al. 2016] search algorithm use this approach. The problem with enumerating fully concrete candidates is that this approach explodes when forced to generate constants because, obviously, there are 2^{64} different choices for a 64-bit constant. A practical compromise is to choose constants from a pre-generated pool, but this will miss synthesis results that require constants from outside the pool. (LENS mitigates this problem by allowing several different synthesis sub-algorithms to collaborate, and some of them are not subject to this problem.)

Souper's design point, depicted as the second one from the right in Figure 2, represents constants and uninstantiated instructions symbolically. When a partially-symbolic candidate cannot be pruned, Souper synthesizes constants using a CEGIS loop; uninstantiated instructions are enumerated one by one.

A more symbolic synthesis technique than Souper's could use CEGIS to wire available values from the specification to inputs in the synthesized code. For example, if a specification has 32 available SSA values, Souper must currently enumerate 960 different ways to connect these values to x and y before the candidate x/y can be rejected.[1] Alternatively, a CEGIS loop could be used to synthesize two 5-bit values, one selecting which input connects to x and the other selecting which input connects to y. This as-yet-untried design point is shown in the middle of Figure 2.

A full CEGIS loop [Gulwani et al. 2011] synthesizes constants, connections to inputs, and also connections between instructions in the candidate all in the same loop. Souper had previously used this strategy, but it turned out to be very difficult to get it to scale up to even modest numbers of instructions. Problems included:

- The size of the query issued by CEGIS scales quadratically with the number of available components, causing many solver timeouts in realistic use cases.
- Synthesizing large constants is a difficult problem by itself, and attempting to synthesize both instructions and (possibly multiple) constants at the same time does not work for some optimizations that we know should be reachable by Souper's search.

[1]The number is not 1024 since any candidate of the form z/z can be rejected syntactically.

Finally, corresponding to the left-most design point in Figure 2, a synthesis problem can be posed to an SMT solver using a single exists-forall query. If f is the specification and the synthesized function g takes a single input x, the query is:

$$\exists g. \forall x. f(x) = g(x)$$

In other words, does there exist a synthesized function that, for every valuation of the input, produces the same result as the specification? There is promising research in this direction [Barbosa et al. 2019].

2.3 Pruning Without Dataflow Analysis

The major problem with enumerative synthesis is that the number of candidates is an exponential function of the amount of synthesized code. Souper attempts to tame the explosion using four techniques that are not contributions of this paper:

- Instead of testing one candidate for each possible choice of value for synthesized constants, leave constants as symbolic and then use the solver to synthesize constants directly.
- Use a cost model to prune branches that cannot lead to candidates that are cheaper than the specification.
- Apply a variety of ad hoc pruning strategies, including: never attempting to shift a one-bit value, exploiting commutativity, not generating instructions with all constant operands, etc.[2]
- Attempt to prune a concrete candidate (one lacking symbolic constants and holes) by interpreting the candidate on several test inputs, rejecting it as soon as its behavior deviates from the behavior of the specification. Previous superoptimizers have made this kind of inequivalence test fast by running native instructions in a sandbox [Bansal and Aiken 2006] and by running checks in parallel [Schkufza et al. 2013].

Although these methods, together, greatly reduce the number of candidates that must be sent to an SMT solver, synthesis remained painfully slow. This is the motivation for our work.

2.4 A Souper Example

Consider the specification $f(x, y) = (x \,\&\, y) + (x \oplus y) - x$. Here, x and y are bitvectors; in practice they would often be 32 or 64 bits, but the specific length does not matter here. $\&$ is the bitwise AND operation and $\oplus$ is the bitwise exclusive-or operation. $+$ and $-$ are bitvector addition and subtraction operations, respectively.

Souper's goal is to synthesize a better way to compute this specification and then, if successful, apply the new optimization while participating in LLVM's middle-end optimization pipeline. It does this by enumerating candidates roughly in order of increasing cost, using a simple platform-independent cost model. The first candidate Souper tries, a symbolic constant C, cannot be synthesized using a CEGIS loop, because there does not exist a concrete value for C such that $f(x, y) = C$. Other early guesses include subexpressions of f such as x and $x \oplus y$. A large proportion of fully concrete candidates, such as these, can be pruned using an interpreter. For example, let $g(x, y) = x \oplus y$. Then, $f(-1, 1) = 0$ but $g(-1, 1) = -2$. This is a proof of inequivalence.

Once candidates with more than one new instruction start to be enumerated, they will contain holes. For example, $g(x, y) = x * H$. If this partially symbolic candidate can be pruned, then it does not need to be expanded into a potentially huge number of hole-free candidates.

[2]We developed a pleasing method for finding opportunities for ad hoc pruning: we run Souper on its own lists of candidates. When a candidate can be optimized to a simpler form, we try to find a simple, syntactical way to avoid generating that candidate in the first place.

In the end, Souper finds 39 candidates that refine f and execute fewer operations, including $y \mathbin{\&} (x \oplus y)$ and $(x \mid y) - x$. Deciding which candidate is preferable requires a detailed, target-aware cost model. (Souper has such a model, but it does not play a role in this work.)

3 RAPIDLY REJECTING SYNTHESIS CANDIDATES USING DATAFLOW ANALYSES

Enumerative program synthesis can be slow because many candidates must be checked, and also because each refinement check requires potentially expensive interactions with an SMT solver. This section is about how we can use dataflow analyses to reduce the number of candidates to be enumerated and avoid some of these SMT queries. As a simple example, consider the specification:

$$f(x) = (x * x * x) \mid 1$$

and the partially symbolic candidate:

$$g(x) = H(x) \ll C$$

Here, H is a hole representing a DAG of arbitrary, not-yet-enumerated instructions whose result is shifted left by a constant C number of bit positions. If this candidate can be pruned, then the lazy enumeration strategy pays off, since filling the hole with concrete instructions would entail evaluating a potentially large number of additional candidates.

A dataflow analysis can conclude that the output of the specification is always odd, or equivalently that its lowest bit is always set: this is a trivial consequence of the bitwise OR operation at the root of the expression tree. Similarly, since it is useless to synthesize a shift by zero bit positions, an analysis can conclude that the output of the candidate is always even, or equivalently its lowest bit is always cleared.[3] The two dataflow facts *conflict*: the intersection of their concretization sets is empty. This fact serves as a proof that no concrete expansion of this partially symbolic candidate can refine the specification. This branch of the search tree can be pruned. The rest of this section explains this pruning method, and others like it, in detail.

3.1 Conventions and Notation

This paper is about loop-free functions over bitvectors. These functions are comprised of primitive operations that correspond to instructions and intrinsic functions in LLVM IR.[4] By convention, we'll refer to the specification as f and the current candidate as g, both of which (unless otherwise specified) take a single bitvector argument and return a single bitvector. Since any bitvector function taking multiple arguments can be trivially rewritten as an equivalent function taking a single argument that is the concatenation of the original arguments, no generality is lost. This convention is solely for notational convenience—our implementation handles functions with multiple arguments.

When we specialize a specification or candidate with a particular choice of input, we'll refer to the input as I. When a candidate contains a symbolic constant, the symbolic constant will be called C, and a hole will be called H. All functions under consideration are pure: they have no state and produce no side-effects.

We will refer to abstract interpreters and dataflow analyses interchangeably. Any such analyzer maps a function to an abstract value. A non-specific abstract interpreter will be called $\mathbb{A}$; specific analyzers will be represented by other letters in the same font. The concretization function γ maps an abstract value to the set of concrete elements that it represents.

[3]The fact that $C > 0$ where C is a shift exponent could not be inferred by a general-purpose static analyzer. Our static analyzers have incorporated a few synthesis-specific facts like this in order to generate results that are more precise.
[4]https://llvm.org/docs/LangRef.html

When it seems clearer, we will write binary numbers like this: 1011_2. A particular bit of a bitvector is referenced using square braces; for example, if $f(x)$ returns a 32-bit value, its least significant bit (LSB) is $f(x)[0]$ and its most significant bit (MSB) is $f(x)[31]$.

3.2 Pruning by Finding Conflicts between Dataflow Facts

The pruning problem is: Given a specification $f(x)$ and a partially symbolic candidate $g(x)$, containing symbolic constants, holes, or both, can we rapidly prove that there does not exist a concrete instantiation of g that refines f?

We attack this problem by attempting to find an abstract fact that is true about f which *conflicts* with an abstract fact that is true about all concrete instantiations of g. The intuition behind a conflict is that it corresponds to an empty intersection between the sets of concrete elements represented by the abstract facts. We look for two kinds of conflict: root and leaf.

Finding root conflicts using forward reasoning. The root of a specification or candidate is its output value. Suppose $\mathbb{A}$ is an abstract interpreter for bitvector expressions. Then, $\mathbb{A}(f)$ is an abstract value that overapproximates the concrete values that f can produce and $\mathbb{A}(g)$ is an abstract value that overapproximates the concrete values that all possible instantiations of g can produce. A *root conflict* exists when

$$\gamma(\mathbb{A}(f)) \cap \gamma(\mathbb{A}(g)) = \emptyset$$

We look for root conflicts using three forward dataflow analyses:

- *Known bits* ($\mathbb{K}$): attempts to prove that each output bit is always either 0 or 1
- *Integer ranges* ($\mathbb{R}$): attempts to prove that the output is within a range of integer values
- *Bivalent bits* ($\mathbb{V}$): attempts to prove that each output bit can be flipped by choosing two different input values

Finding leaf conflicts using backward reasoning. The leaves of a candidate are input values, symbolic constants, and holes. To find leaf conflicts, we use three backward dataflow analyses to compute properties of leaves:

- *Required bits* ($\mathbb{X}$): attempts to prove that individual input bits influence the output
- *Don't-care bits* ($\mathbb{D}$): attempts to prove that individual input bits do not influence the output
- *Forced bits* ($\mathbb{F}$): tracks individual bits in symbolic constants which are forced to be a particular value, given a specific output value

Table 1 summarizes pruning using these analyses.

Better pruning using input specialization. We found that, in practice, pruning using a straightforward application of dataflow analysis often failed, because many specifications and candidates are simply not very susceptible to static analyses such as known bits. A solution is suggested by the fact that if we create specialized versions of $f(x)$ and $g(x)$ for a particular value of x, and find a conflict between these, then we have shown a conflict overall. Since specifications are fully concrete, specializing $f(x)$ results in a concrete value: no abstract interpretation is required.

Pruning in the presence of path conditions. Adding a path condition (introduced in Section 2.1) to a specification can result in more candidates being valid. In other words, path conditions are often a necessary part of the justification for optimizations derived by Souper. About a third of the synthesis candidates extracted by Souper while compiling SPEC CPU 2017 have path conditions. The flip side of this is that by making optimizations easier, path conditions make pruning more difficult.

Table 1. Summary of dataflow-based pruning methods described in the later sections

(R)oot or (L)eaf?	Analysis of Specification	Analysis of Candidate	Pruning Condition	Paper Section
R	$X = f(I)$	$Y = \mathbb{K}(g(I))$	$X \notin \gamma(Y)$	3.3
R	$X = f(I)$	$Y = \mathbb{R}(g(I))$	$X \notin \gamma(Y)$	3.4
R	$X = \mathbb{K}(f)$	$Y = \mathbb{V}(g)$	$\exists i.(X[i] \neq \top) \wedge (Y[i] = 1)$	3.5
L	$X = \mathbb{X}(f)$	$Y = \mathbb{D}(g)$	$\exists i.(X[i] = 1) \wedge (Y[i] = 1)$	3.6
L	$X_1 = f(I_1)$, $X_2 = f(I_2)$	$Y_1 = \mathbb{F}(g(X_1))$, $Y_2 = \mathbb{F}(g(X_2))$	$\gamma(Y_1) \cap \gamma(Y_2) = \emptyset$	3.7

For example, if $f(x) = x \,\&\, 7$ and $g(x) = x$, g obviously does not refine f, and we would like to prune g without consulting a solver. However, if f is qualified by a path condition $x < 8$, then g refines f. In this case, it would obviously be a mistake to prune g.

The basic criterion for a root conflict is $\gamma(\mathbb{A}(f)) \cap \gamma(\mathbb{A}(g)) = \emptyset$. In other words, the specification and candidate produce completely disjoint outputs. This is a very strong condition; if it holds overall, then it still holds in the presence of an arbitrary path condition, because a path conditions can only reduce the number of behaviors exhibited by a specification, not increase it.

For our pruning methods where the specification and candidate are specialized using an input value, we must be careful that the input value respects the path conditions. We accomplish this by executing the path condition on the inputs for specialization, throwing away those that fail to meet the condition. This does not work well when a path condition is extremely specific, such as requiring a 64-bit value to be in the range 100000..100007. In that case we could make solver queries to get inputs satisfying the path conditions, but since the goal of pruning is to avoid solver queries, we do not do this, and rather allow pruning to fail.

Pruning in the presence of undefined behavior. Formal reasoning about LLVM code requires dealing with three kinds of undefined behavior [Lee et al. 2017]:

- *undef* values that can evaluate to any value representable by a given data type,
- *poison* values that are contagious and live outside of the normal value space, and
- *immediate undefined behavior* which destroys the meaning of an LLVM program in the same way that undefined behaviors in C and C++ are semantically catastrophic.

The top-level consequence is that Souper's candidates are not checked for equivalence with the specification, but rather for refinement of the specification. Basically, the candidate is permitted to be more defined than the specification, but not less. In the absence of undefined behavior, refinement degenerates to simple equivalence.

For the most part, undefined behavior plays only a small role in our pruning strategies, but we do have to be careful about two important things. First, when interpreting concrete Souper expressions (which follow the semantics of the corresponding LLVM expressions), we track the introduction of poison values and undefined behavior. For example, if `INT_MAX` is incremented using LLVM's `add nsw` instruction, the result is poison, and if a number is divided by zero, the result is immediate undefined behavior. We never encounter explicit undef values during interpretation. If specializing a specification with a concrete input leads to poison or undefined behavior, we simply drop that input. If specializing a candidate with a concrete input leads to poison or undefined behavior (and specializing the specification did not) then this is by definition a failure of refinement and we can prune the candidate.

The second area where we have to be careful about undefined behavior is in our dataflow analyses. Here we adopt the same strategy that is already used to good effect inside LLVM: dataflow analyses return an abstract value which overapproximates the set of values which can be produced without triggering undefined behavior. In other words, the reasoning that allows pruning to reject candidates is based on conflicts between defined executions.

3.3 Finding Root Conflicts using Input Specialization and Known Bits

Known bits is a forward dataflow analysis that attempts to prove that each bit of the result of a function is either zero or one, or else conservatively returning $\top$. A maximally precise known bits analysis is defined as:

Given a function $f(x)$, $\mathbb{K}(f(x))[i]$ is:

- 0 if $\forall x. f(x)[i] = 0$
- 1 if $\forall x. f(x)[i] = 1$
- $\top$ otherwise

For example, if a 32-bit value is shifted left by 10 bit positions, the known bits analysis can conclude that the result has zeroes in bit positions 0–9. Practical implementations of known bits will typically fall short of maximal precision by sometimes returning $\top$ in situations where they could have returned 0 or 1. The concretization function, γ, explodes every unknown bit into both possibilities:

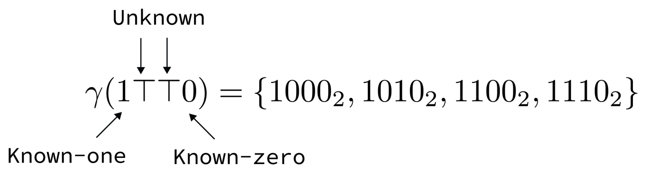

Fig. 3. Known Bits concretization

Conflict criteria. We use the known bits analysis to prove inequivalence through a root conflict. A conflict appears when a bit is known to be zero in the specification and known to be one in the candidate, or vice versa. This makes the intersection of the concretization sets empty, which is formalized as:

$$\gamma(\mathbb{K}(f)) \cap \gamma(\mathbb{K}(g)) = \emptyset \rightarrow f \neq g$$

Consider this specification:

$$f(x) = x^2 + 1$$

and a partially symbolic candidate:

$$g(x) = x \;\&\; H(x)$$

Alas, there is too much uncertainty in these functions for abstract interpretation to succeed. Even using a maximally precise analysis, both f and g return $\top$ in all bit positions. However, specializing

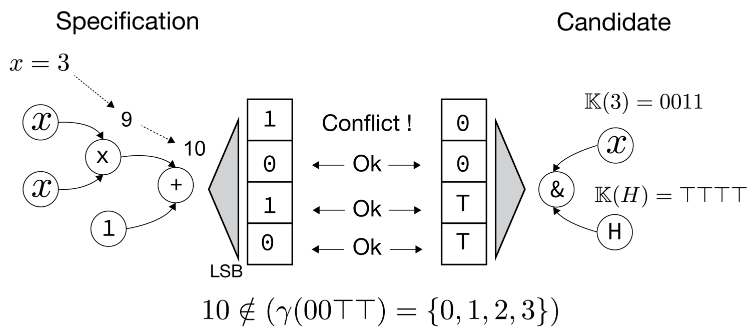

Fig. 4. Finding a root conflict between $f(x) = x^2 + 1$ and $g(x) = x \mathbin{\&} H$ by combining input specialization and the known bits analysis

f and g with a concrete input allows the abstract interpreter to return a non-$\top$ result for at least some bits. Given a concrete input value I, the pruning criterion becomes:

$$f(I) \notin \gamma(\mathbb{K}(g(I))) \rightarrow f \neq g$$

For example, if $I = 3$ and f and g return 4-bit results, then $f(0011_2) = 1010_2$ and $\mathbb{K}(g(0011_2)) = 00\top\top$. Next, we observe that $\gamma(00\top\top) = \{0, 1, 2, 3\}$. Since 10 ($1010_2$) is not a member of this set, we have found a root conflict and g can be pruned. Thus, the hole does not need to be expanded and this subtree of the enumerative search space can be pruned away. Figure 4 illustrates this example.

The case where a conflict can be found only using input specialization is the common case in practice. Furthermore, if specialization using one input value fails to find a conflict, a different value may succeed. This leaves the number of specialization attempts, and how input values are chosen, as engineering tradeoffs (discussed in Section 4).

Abstract transfer functions for bit-level operations in the known bits domain tend to be efficient and precise. Operations such as addition and subtraction in this domain are more difficult, and multiplication and division are even harder.

3.4 Finding Root Conflicts using Input Specialization and Integer Ranges

The integer range analysis is another forward analysis. Pruning using integer ranges works in much the same way that pruning using known bits does.

We reused the integer range abstract domain from LLVM's ConstantRange class, which allows integer ranges to wrap across both the maximum two's complement and unsigned boundaries. Inconveniently, this domain does not form a lattice, so it does not admit a straightforward definition of a maximally precise transfer function. However, all correct transfer functions will satisfy this definition:

Given a function $f(x)$, $\mathbb{R}(f(x))$ is $[A, B)$ or $\top$ or $\bot$ where:

- if $B > A$: $\forall x. f(x) \geq A \land f(x) < B$
- if $B < A$: $\forall x. f(x) \geq A \lor f(x) < B$
- $\top$ is always a sound result
- $\bot$ is only a sound result for undefined behavior

Comparison operations in this definition are all unsigned.

For example, $[1, 4)$ concretizes to $\{1, 2, 3\}$ and the range $[10, 2)$ concretizes to $\{x | x \geq 10 \vee x < 2\}$. Of course, in general we prefer transfer functions that return abstract values whose concretization sets are as small as possible. $\perp$ can be returned when analyzing an expression such as $x/0$.

Conflict criteria. We show integer range conflicts in the same way that we show known bits conflicts. In other words, if the output ranges of the specification and a candidate are disjoint, that candidate cannot be a valid refinement of the specification. This criterion is:

$$\gamma(\mathbb{R}(f)) \cap \gamma(\mathbb{R}(g)) = \emptyset \rightarrow f \neq g$$

Again, the unspecialized pruning condition does not work very often and we improve its efficacy by specializing the specification and candidate using concrete inputs. Given a concrete input I, the pruning criterion is then:

$$f(I) \notin \gamma(\mathbb{R}(g(I))) \rightarrow f \neq g$$

3.5 Finding Root Conflicts using Known Bits and Bivalent Bits

We created a new abstract domain, *bivalent bits*, which identifies the bits in the output of a partially symbolic candidate that provably hold both possible values. This domain is a dual of known bits that approximates in the other direction.

Bit i of the output of a concrete function $f(x)$ is *bivalent* if:

$$\exists x, y. f(x)[i] \neq f(y)[i]$$

Otherwise, the bit is $\top$.

We are primarily concerned with finding the bivalent bits of partially symbolic functions. Crucially, a bit is only bivalent if it is bivalent for every value of symbolic constants and for every instantiation of holes. The definition given above can be extended to apply to all concrete instantiations of the given function.

Let G be the set of all concrete instantiations of a partially symbolic function $g(x)$. Bit i of g is bivalent if:

$$\forall g' \in G. \exists x, y. g'(x)[i] \neq g'(y)[i]$$

Otherwise, the bit is $\top$.

All bits of the input to a function are bivalent, by definition. No bit of a constant, either symbolic or concrete, is bivalent. A somewhat tricky issue in writing this analysis was taking correlated values into account. We did this in a conservative fashion: we consider an expression non-bivalent if it has two subexpressions that use a common variable. Thus, the output bits of both $g_1(x) = x - x$ and $g_2(x) = x + x$ are considered to be non-bivalent. The result returned for g_1 is sound and precise, whereas the result for g_2 is sound but not precise. A less conservative bivalency analysis could conclude that all but the bottom bit of g_2 are bivalent.

Conflict criterion. If a bit is known in the specification and bivalent in the candidate, the candidate can be pruned. For this pruning method, we do not specialize the candidate with an input value: we analyze it in its general context. The conflict criterion is formalized as:

$$(\exists i. \mathbb{K}(f)[i] \neq \top \wedge \mathbb{V}(g)[i] \neq \top) \rightarrow f \neq g$$

Example. Consider the specification $f(x) = (x + 2) \mid 1$ and the partially symbolic candidate $g(x) = x + C$. The LSB of $f(x)$ can be shown to always be 1, no matter what the input is, by the known bits analysis. The LSB of $g(x)$ is bivalent because, for all values of C, $x = -C$ makes the LSB of $g(x)$ to be 0 and $x = 1 - C$ makes it 1. Thus, no instantiation for $g(x)$ can result in a valid refinement; this candidate can be pruned. This pruning technique is particularly useful for checking synthesis candidates with more than one input, and at least one symbolic constant.

Dealing with preconditions. This pruning technique requires two analyses, known bits and bivalent bits, to be performed without input specialization. This is because specialization makes all input bits non-bivalent. For known bits without input specialization, all the input variables are considered unknown, or $\top$. Similarly, for bivalent bits the non-correlated uses of input variables are considered fully bivalent. This strategy fails to work in the presence of a precondition.

To solve this problem, we start with a set of inputs that satisfy the precondition and then use these abstract domains' respective abstraction functions to lift the sets of concrete values into the abstract domains. This function, α, satisfies $\alpha(\gamma(x)) = x$ where x is an arbitrary abstract value.

For example, given a precondition $P(x) = x > 0$ and $x < 4$, we choose an input set like $\{1, 3\}$ which satisfies the precondition. Assuming 4-bit numbers, we obtain the following results via abstraction:

$$\alpha_{\mathbb{K}}(\{0001_2, 0011_2\}) = 00\top 1$$
$$\alpha_{\mathbb{V}}(\{0001_2, 0011_2\}) = \top\top B\top$$

In the above example, B is the abstract value indicating a bivalent bit. Now, if we had to prove two functions f and g inequivalent (given the precondition P), the known bits abstract interpreter would use $00\top 1$ as the input abstract value for analyzing f, and the bivalent bits abstract interpreter would use $\top\top B\top$ as the input abstract value for analyzing g.

This technique works only because the abstraction function for the bivalent bits domain produces an underapproximation of the precondition. In contrast, known bits abstraction produces an overapproximation. Overapproximation of the precondition for both of the functions can lead to incorrect pruning.

3.6 Finding Leaf Conflicts using Required Bits and Don't-Care Bits

This pruning method is not novel: it was developed for McSYNTH [Srinivasan et al. 2016]. The idea is that $f(x, y) = x + y$ and $g(x, y) = x + C$ can be proved inequivalent by observing that g does not use y, whereas y is a *required* input for f. A variable is required if changing its value influences the output. For example, only the four least significant bits of I in $g(I) = I \,\&\, 0xF$ are required. This analysis is a backward analysis, and is defined:

Given $f(x)$, bit i of x is *required* if:
$\exists m. f(m) \neq f(m')$,
where m' is m with bit i flipped.
Otherwise, the bit is $\top$.

Similarly, an input bit is considered *don't care* if it cannot influence the output of a function. This is also a backward analysis:

Given $f(x)$, bit i of x is *don't care* if:
$\forall m. f(m) = f(m')$,
where m' is m with bit i flipped.
Otherwise, the bit is $\top$.

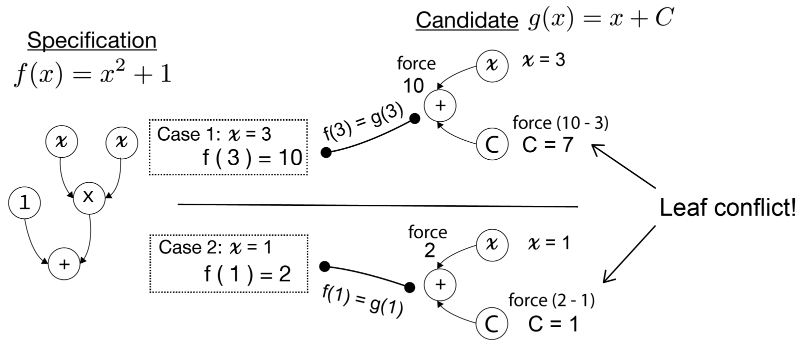

Fig. 5. Finding a leaf conflict using forced bits

Conflict criterion. A specification and a candidate are inequivalent if any bit of an input variable is *required* in the specification and *don't care* in the candidate. Given a set of input variables X, f is inequivalent to g if:

$$\exists i, x \in X.(\mathbb{X}(f, x)[i] \neq \top) \wedge (\mathbb{D}(g(x))[i] \neq \top)$$

For example, assume that $f(x, y) = x * y + 42$ and $g(x, y) = x^2 + C$. Here, all the bits of x and y in f are required but only x is used in g. Hence, g cannot be equivalent to (or a refinement of) f. This technique is only useful for partially symbolic candidates without holes.

Dealing with path conditions. A path condition can reduce the number of required bits of a function. For example, in the function $f(x) = x$, all the bits are required but given a condition $P(x) = x < 16$, only the 4 least significant bits of x are required. Hence, the definition of required bits has to be adapted to account for a path condition. This is given as follows:

Given f and a precondition P, bit i of the input to f is *required* if:
$\exists m.P(m) \wedge f(m) \neq f(m')$,
where m' is m with bit i flipped.

On the other hand, path conditions make a don't care bits result imprecise but not unsound. Hence, the original definition still works and the following one can be used if increased precision is desirable:

Given f and a precondition P, bit i of the input to f is *don't care* if:
$\forall m.P(m) \longrightarrow f(m) = f(m')$,
where m' is m with bit i flipped.

3.7 Finding Leaf Conflicts using Forced Bits

Our final pruning technique uses a *forced bits* backward analysis to find a conflict in a symbolic constant. Basically, we show that the constant must take on one value to make the candidate work for one concrete input, and another value to make the candidate work for a different input.

Consider the specification

$$f(x) = x^2 + 1$$

and the partially symbolic candidate

$$g(x) = x + C$$

In this example, $f(x)$ evaluates to 10 for $x = 3$, and 2 for $x = 1$. For an instantiation of g to be a valid refinement of f, there must be a value of C for which the two following equations are satisfiable.

- $f(3) = g(3) = 10 = 3 + C$
- $f(1) = g(1) = 2 = 1 + C$

The first equation forces C to be 7 and the second one forces C to be 1. The system of equations is unsatisfiable, giving us a leaf conflict that justifies pruning g without instantiating C or invoking a solver. Figure 5 illustrates this example.

Solving for the unique value of a symbolic constant given a function's output means that we need to be able to invert operations that make up the function. This is not difficult for the addition in g above, but it is more difficult for bitwise operations. For example, $10_2 \ \& \ C = 00_2$ does not force C to have a unique value. However, the MSB of C has to be 0; we call this a *forced bit* result. A forced bit conflict is similar to a known-bits conflict: $0\top$ conflicts with $1\top$ but not with $\top\top$. Formally:

Given a partially symbolic function $g(x)$ containing a symbolic constant C, and a known result R, bit i of $\mathbb{F}(g, C, R)$ is:

- 0 if $\forall C.C[i] = 1 \longrightarrow \nexists x.g(x) = R$
- 1 if $\forall C.C[i] = 0 \longrightarrow \nexists x.g(x) = R$
- $\bot$ if both of the above are true (conflict)
- $\top$ otherwise

Intuitively, this means that if a bit of a symbolic constant is forced to be zero, there is no way to equate the result of the function with a given known result when that bit is one, and vice versa for forced-one. This analysis is used to find leaf conflicts in candidates with symbolic constants. A leaf conflict occurs if the only possible concrete values of a symbolic constant come from two mutually exclusive sets when the candidate is specialized with different inputs. The concretization function for this abstract domain is the same as known-bits, and this analysis can be considered a backward known-bits analysis.

The pruning condition for the forced bits analysis is given as follows. Given a symbolic constant C, and two inputs x_1 and x_2:

$$\gamma(\mathbb{F}(g, C, f(x_1))) \cap \gamma(\mathbb{F}(g, C, f(x_2))) = \emptyset$$

4 IMPLEMENTATION

Prior to our work, Souper had an enumerative synthesis procedure that tamed the number of candidates using partially symbolic candidates, cost-based pruning, ad hoc pruning methods, and input specialization for concrete candidates. We added a modular framework for pruning that calls out to pluggable pruning methods from various points within the enumerative synthesizer. Our pruning and abstract interpretation code is about 3,500 lines of C++. The overall method is illustrated in Algorithm 1.

Dataflow analyses. The basis for our new pruning work is a collection of dataflow analyses. For one of these, the integer range analysis, we were able to directly reuse existing LLVM code, where the ConstantRange class provides a collection of transfer functions for this abstract domain.[5]

[5] https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/IR/ConstantRange.cpp

Algorithm 1 Dataflow pruning

```
1: procedure DATAFLOW-PRUNING($f, g, Inputs$)
2:     known = $\mathbb{K}$(f)
3:     restricted = known.zero | known.one
4:     if $\mathbb{V}(g)$ & $restricted \neq 0$ then
5:         Return Success.                    ▷ Root conflict – known in $f$ but bivalent in $g$
6:     if g does not have a $Hole$ then
7:         for $x$ in $params(f)$ do                        ▷ Each parameter is a leaf
8:             if $(\mathbb{X}(f, x)$ & $\mathbb{D}(g, x)) \neq 0$ then
9:                 Return Success.      ▷ Leaf conflict – $x$ is required in $f$ but don't care in $g$
10:    forced-bits = $\emptyset$
11:    for I in Inputs do
12:        X = f(I)
13:        if $X \notin \gamma(\mathbb{K}(g, I))$ then
14:            Return Success.                ▷ Root conflict – proved by known bits analysis
15:        if $X \notin \gamma(\mathbb{R}(g, I))$ then
16:            Return Success.                ▷ Root conflict – proved by integer ranges analysis
17:        forced-bits $\xleftarrow{append}$ $\mathbb{F}(g, X)$
18:        if find-conflict(forced-bits) then
19:            Return Success.                ▷ Leaf conflict – proved by forced bits analysis
20:    Return Failure.                                   ▷ ($g$ could not be pruned.)
```

Unfortunately, LLVM's known bits analysis is not engineered in such a way that it can be reused directly. We reused (via copy and paste) LLVM's known bits transfer functions for addition and subtraction, and implemented the rest of this static analysis from scratch. We wrote the other four static analyzers in our pruning framework (bivalent bits, required bits, forced bits, and don't-care bits) from scratch as well.

In some cases, constants synthesized by Souper have a restricted range. For example, a constant used for a shift exponent will always be greater than zero and less then the bitwidth of the value being shifted. Our dataflow analyzers take these constraints—which would not be valid in general, for LLVM IR—into account whenever possible to increase precision.

Speed vs. precision. Every static analyzer must make tradeoffs between precision and runtime performance. In this work, we generally preferred making our static analyzers fast rather than being extremely precise. We did spend a fair amount of time investigating ways to increase the precision of our dataflow analyses, but we found that in many cases, such efforts slowed down Souper's synthesis overall, because the resulting increase in the success rate of pruning was not enough to offset the increased execution time of the more precise static analyzers. We believe there is room for improvement in striking a principled balance between precision and performance here.

Inputs for specialization. The overhead of a pruning method that uses input specialization grows linearly with the number of inputs. It is important to choose an appropriate number of inputs, and also to choose the specific input values properly. We investigated using randomly chosen inputs as well as hand-picked values such as 0, 1, and −1 that seemed likely to work well for specialization. Figure 6 shows that:

- Pruning effectiveness flattens out before 10 inputs for both strategies.

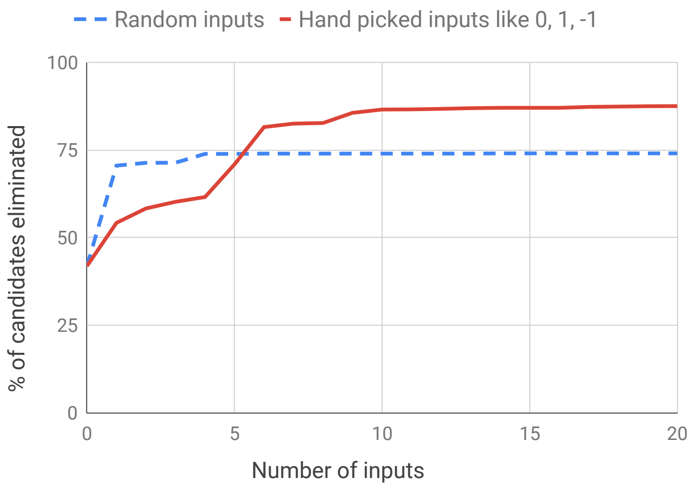

Fig. 6. The y-axis tracks the percentage of synthesis candidates which did not require an SMT solver to reason about. This figure shows that increasing the number of inputs beyond about 10 has diminishing returns.

- Choosing values by hand is more effective than choosing values at random.

We chose a hybrid approach that combines 15 chosen and random inputs; it performs marginally better than the solid line in Figure 6.

Pruning for pruning. In certain situations, pruning cannot succeed. For example consider any specification and the partially symbolic candidate $g(x) = H_1 + H_2$. H_1 and H_2 are two holes that can be instantiated with any pair of functions. Since this candidate lacks any information that might lead to successful pruning, we would prefer to avoid wasting time attempting to prune it.

We developed a simple, very fast static analysis to conservatively identify candidates that cannot be pruned. The question answered by the transfer functions for this analysis is: Given an operation with several inputs, which inputs need to be holes (or *indeterminate*) for the result to be indeterminate. These transfer functions turned out to be tricky to write and debug. We solved this problem offline using an SMT solver. The transfer functions are coarse and only reason about the connection between operations; they do not look at values at all.

Given a binary operation Op that takes one of its inputs from a hole H, we want to find out if pruning could possibly succeed. It cannot succeed (that is, we can prune the pruner) if this formula is satisfiable:

$$\exists H, y. \forall x, z. y \neq Op(H(x, z), z)$$

This formula asks if there exists an arbitrary function H for which there is an unachievable output y with all possible inputs x, z. This is not a first order formula, but it can be translated into an equisatisfiable first order formula using Ackermannization—replacing H with a fresh bitvector (the general case of Ackermannization is a bit more complex than this [Bruttomesso et al. 2006]).

For example, consider bitwise AND. The output $y = 1$ cannot be produced when the hole returns a 0, no matter what the other input z is. The same statement is not true for two's complement addition, where no matter what is returned by the hole, it is possible to choose the other input to

Algorithm 2 Exhaustively computing the precise set of results that an operation in the known bits abstract domain should return

```
procedure EXHAUSTIVE-KNOWN-BITS($A$, $B$, Op)
    if $A$ is not concrete then
        $result_1$ = exhaustive-known-bits(set-lowest-unknown($A$), $B$)
        $result_2$ = exhaustive-known-bits(clear-lowest-unknown($A$), $B$)
        return $result_1 \cup result_2$
    if $B$ is not concrete then
        $result_1$ = exhaustive-known-bits($A$, set-lowest-unknown($B$))
        $result_2$ = exhaustive-known-bits($A$, clear-lowest-unknown($B$))
        return $result_1 \cup result_2$
    if $\mathrm{Op}(A, B)$ is undefined then
        return the empty set
    return $\mathrm{Op}(A, B)$
```

produce any arbitrary output. Thus, the output of the addition operator is fully indeterminate, and pruning cannot succeed.

This analysis is performed on each partially symbolic candidate with a hole. It reduced the total time spent in pruning by 30% without having a functional effect on the results of the superoptimizer.

Testing abstract interpreters for correctness. Since our abstract interpreters are parameterized by bitvector size, a relatively easy testing method presented itself: exhaustive testing at small bitwidths. For example, at width w the known bits abstract domain contains 3^w abstract values. Therefore, the known bits transfer function for a binary operation such as addition can be exhaustively tested at four bits by enumerating all 3^4 values for its left operand and all 3^4 values for its right operand, for a total of 6,561 test cases. For each test case we exhaustively compute the maximally precise result by expanding each input using the concretization operator and applying the concrete addition operation to the cross-product of the resulting sets; this is shown in Algorithm 2. Finally, it remains to lift the set of results into the known bits abstract domain and then check that the answer returned by our transfer function is an overapproximation of this maximally precise result.

All of our abstract transfer functions for the known bits and bivalent bits analyses have been validated using this approach. (We trust that the ConstantRange functions we reuse from LLVM are correct; they are heavily used in a production compiler.) With this technique, we found several bugs early in the implementation. We have not yet validated our other static analyzers using this technique.

Testing pruning soundness. We want to ensure that no candidate is improperly pruned. We tested this by running enumerative synthesis over Souper's test suite twice. In the first run, dataflow-based pruning was enabled, and in the second run it was disabled. Any difference in the outcome of these two runs must be due to

- a flaw in a pruning algorithm,
- an unsoundness in one of the abstract interpreters, or
- a timeout or a crash.

This testing method found several bugs during the early development of our framework and, notably, helped teach us about some subtleties in reasoning about pruning in the presence of path conditions.

5 EVALUATION

This section investigates the efficacy of our pruning methods. It does not investigate Souper's effectiveness in generating optimizations for LLVM IR: Souper as a whole is not a contribution of this paper, and such an evaluation is not within the scope of our work.

The top-level result of our work is that when all five dataflow-based pruning techniques are combined, we can speed up a large, realistic synthesis workload by 2.32x. We obtained this workload by compiling the C and C++ programs in SPEC CPU 2017[6] with Clang+LLVM 10.0,[7] loading Souper as an optimization pass. To avoid LLVM-related overhead in our experiments, we did not perform synthesis during the compilation, but only extracted synthesis specifications into a cache. The result of this step was a collection of 269,113 synthesis specifications. Each specification consists of an arbitrary-sized DAG of Souper instructions and optionally includes path conditions. Because the cache de-duplicates, these specifications are globally unique.

Next, we ran synthesis on all of these candidates with a limit of two synthesized instructions. In this setup, synthesis candidates can contain an arbitrary number of instructions, but only two of them will be new ones: the rest are reused from the specification. A large majority of the optimizations found in LLVM's peephole optimizers can be expressed in terms of at most two new instructions.

Individual synthesis problems are not parallelized, and each was limited to 300 seconds of CPU time and 4 GB of RAM. The solver used by Souper was Z3 version 4.8.8,[8] and individual invocations of Z3 (of which there may be many, while solving a single synthesis problem) were timed out after 20 seconds. We ran synthesis problems in parallel on an otherwise idle Linux machine with 256 GB of RAM and two AMD EPYC 7502 processors; each of these processors has 32 cores and 64 threads, and is clocked at 2.5 GHz.

Table 2. Comparing synthesis with and without pruning for 269,113 specifications from the C and C++ programs in SPEC CPU 2017. We count synthesis as successful when at least one candidate is found that has lower cost than the specification. A synthesis attempt is killed when it runs out of time (300 seconds) or memory (4 GB).

	Total Time	Successes	Killed
Without pruning	33.76 hours	10095	36757
With pruning	14.54 hours	10892	21682

Table 2 summarizes the results of this experiment. Overall, about 4% of synthesis attempts succeed in the sense of arriving at a candidate that improves upon the specification. Enabling pruning makes synthesis 2.32x faster while at the same time increasing the number of synthesis successes by 8%. The number of successes increases because, in some cases, enabling pruning makes the difference between completing before or after the timeout. Enabling pruning reduced the number of timeouts by 41%. When enabled, the pruning procedure accounts for less than 3% of the run time of Souper.

Figure 7 shows a CDF of synthesis successes with and without pruning, as a function of time. The "with pruning" line does not quite reach 1.0 because there are some cases where pruning slows synthesis down, allowing synthesis without pruning to succeed when it fails with pruning. The largest gap between the with-pruning and without-pruning curves is 13%, occurring at around 10 seconds. In other words, if we set the synthesis timeout to 10 seconds, we would end up with 13%

[6]https://www.spec.org/cpu2017/
[7]https://releases.llvm.org/download.html#10.0.0
[8]https://github.com/Z3Prover/z3/releases/tag/z3-4.8.8

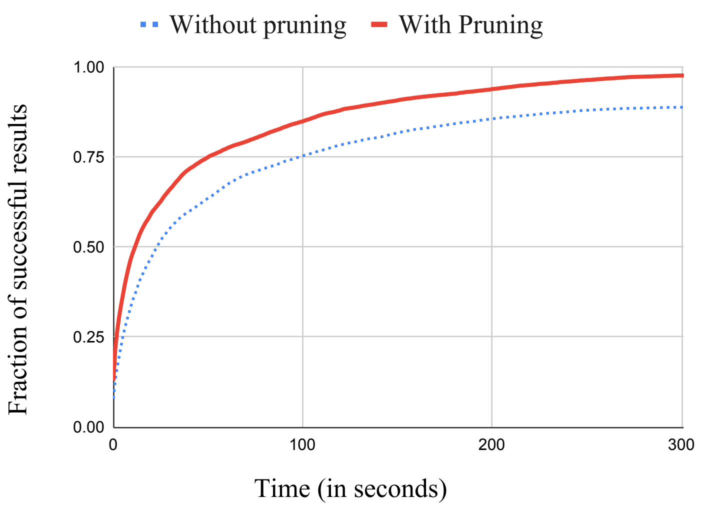

Fig. 7. CDF of time to completion for individual synthesis successes using the specifications from the C and C++ programs in SPEC CPU 2017

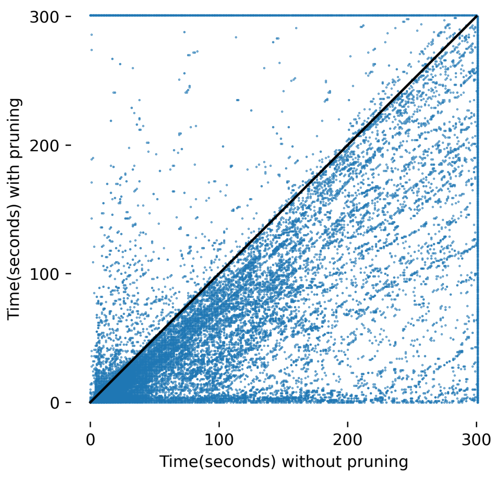

Fig. 8. Comparing CPU time until synthesis success with and without pruning, for specifications from SPEC CPU 2017

more successes due to turning on pruning. Looking at this CDF, it might seem counter-intuitive that pruning speeds up our large synthesis batch job containing the SPEC CPU 2017 specifications by a factor of 2.3x. The large speedup happens because pruning disproportionately speeds up synthesis failures, which are not represented in this graph.

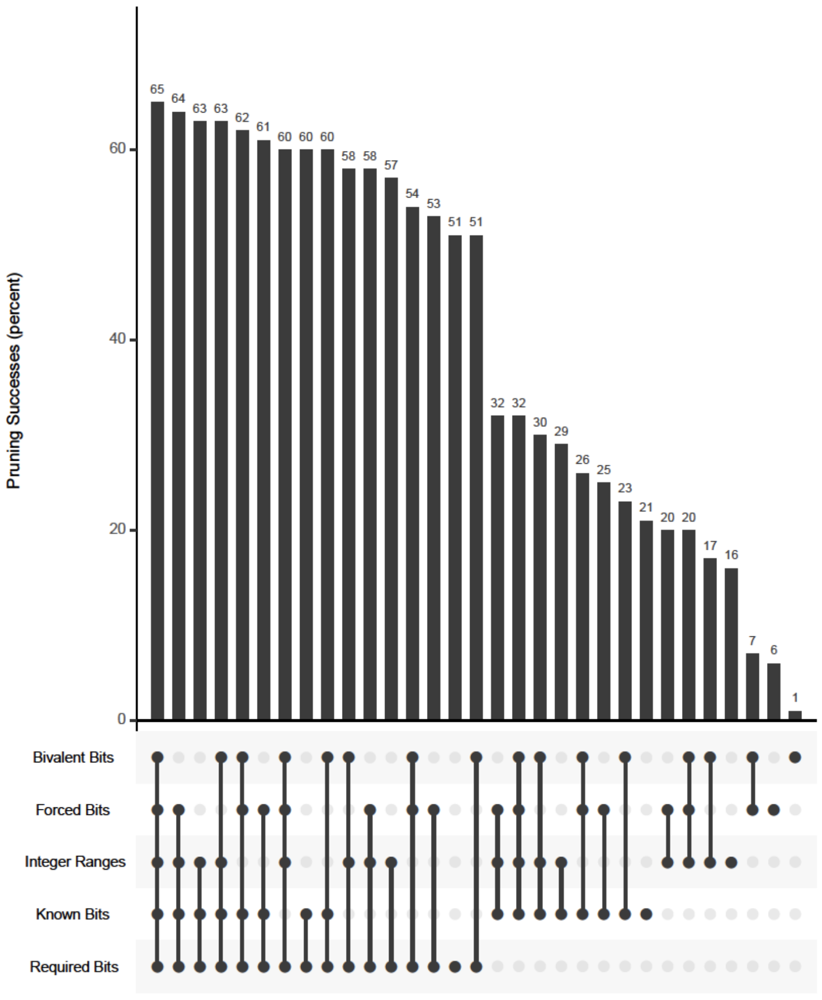

Fig. 9. Comparing the effect of combinations of pruning methods on the number of times Souper must resort to querying an SMT solver

Figure 8 plots, for every synthesis success on the specifications from SPEC CPU 2017, the CPU time taken with pruning against the CPU time taken without pruning. 87% of synthesis successes were obtained more rapidly with pruning. When pruning makes synthesis slower, it is typically for a very large specification containing hundreds of instructions.

Figure 9 shows how the five dataflow-based pruning methods interact.[9] For different combinations of the pruning methods, it measures the reduction in the number of candidates for which Souper needs to invoke the SMT solver. There are two kinds of reduction that we accounted for. First, if a pruning method eliminates a candidate containing one or more symbolic constants—but no holes—then the number of times the solver was required is reduced by one. Second, if a pruning method eliminates a candidate containing at least one hole, then the number of solver invocations is reduced by a larger number. This is because this kind of candidate, if it is not pruned, must be

[9]This graph was produced by UpSet [Lex et al. 2014], a tool that improves upon many-way Venn diagrams.

expanded into a potentially large number of hole-free candidates. For this experiment, pruning of fully concrete candidates using the interpreter approach was always enabled. Here we did not use the full set of specifications from SPEC CPU 2017, but instead used a smaller collection of about 3,000 specifications that we extracted from gzip.

6 RELATED WORK

Enumeration of candidates is a popular strategy for synthesis because it is straightforward to implement and debug, it asks less of the SMT solver than do other synthesis methods, and it admits a great number of customizations such as domain specific and ad-hoc pruning strategies.

Enumerative synthesis has been applied successfully for superoptimization [Bansal and Aiken 2006; Phothilimthana et al. 2016], a protocol specification tool [Udupa et al. 2013], machine code synthesis [Srinivasan and Reps 2015], and data structure synthesis [Loncaric et al. 2018]. There are two ways to scale up enumerative synthesis: reducing the size of the search space and making it faster to evaluate each candidate. Our work has elements of both methods: it reduces the size of the search space by pruning synthesis candidates that contain holes that have not yet been filled with instructions, and it makes evaluating individual candidates faster by using dataflow analyses to avoid potentially-expensive calls into the SMT solver.

The general enumerative synthesis approach used by existing tools (LENS [Phothilimthana et al. 2016], Barthe et al. [2013], CodeHint [Galenson et al. 2014]) can be described as follows. Synthesis candidates are partitioned into equivalence classes based on their behavior on a set of test inputs. A candidate cannot be equivalent to a specification if it produces a wrong output for any one of the test inputs. This is a practical search space pruning technique because the specification and all the corresponding valid candidates are guaranteed to belong to the same equivalence class. Note that producing the same output for all the test inputs (i.e. belonging to the same equivalence class) does not guarantee equivalence. Once a candidate which passes these tests is found, a SAT or SMT solver might be used to determine if there is any input that proves the specification and the candidate to be inequivalent. If such a counterexample is found, it is added to the set of test inputs and the associated equivalence classes are updated. Enumerative synthesis algorithms can be distinguished by how these equivalence classes are constructed and updated. LENS [Phothilimthana et al. 2016] employs a bidirectional search strategy for improving enumerative synthesis. It solves two problems. First, it eliminates restarts; so that once a counterexample is found, the candidates which have been shown to be different (using a recently invalidated set of equivalence classes) should not be evaluated again. Second, it minimizes the number of test cases required by eliminating test cases which result in similar program states, and are thus redundant.

Souper provides a lazy enumeration implementation, which produces partially symbolic candidates containing symbolic constants, holes, or both. To the best of our knowledge, the idea of clustering candidates into equivalence classes has not been shown to work with partially symbolic candidates. The method presented in this paper can be taken as a generalization of testing concrete candidates to testing all candidates: both concrete and partially symbolic. The notion of testing is changed from interpretation to abstract interpretation wherever required. Hence, techniques improving enumerative synthesis are, in general, amenable to be used in combination with our approach.

The most directly related prior work can be found in McSYNTH [Srinivasan et al. 2016], which developed a pruning method that is functionally equivalent to our method from Section 3.6. This approach is effective for pruning both concrete candidates and partially symbolic ones with symbolic constants. It is not generally effective for pruning candidates with holes because holes must be analyzed conservatively by this analysis. Our work builds directly on this foundation and adds four additional dataflow-based pruning techniques.

Several other semantic techniques for pruning the search space for enumerative synthesis have been developed. Sketching [Bornholt et al. 2016; Solar-Lezama 2008] reduces the synthesis search space using a high level program structure called a sketch, with the aim of filling out the holes with a CEGIS procedure. The metasketching paper [Bornholt et al. 2016] describes a cost based pruning technique similar to Souper's [Sasnauskas et al. 2017]. Alur et al. [2017] describe a divide and conquer approach that enumerates over smaller terms and combines them into larger expressions by *unification* [Alur et al. 2015]. This general technique is ideal when there exist structural restrictions on the candidates (as in SyGuS [Alur et al. 2013]). The EUPHONY [Lee et al. 2018] tool contributes another technique to guide the search for this problem. It learns a probabilistic model which guesses the likelihood of correctness of a synthesis candidate. The candidates are enumerated in the order of likelihood guessed by this model.

Looking further back, Bansal and Aiken's superoptimizer [Bansal and Aiken 2006] used concrete execution (in a native sandbox, to avoid interpreter overhead) to disqualify most candidates and then used a SAT solver to verify equivalence for the minority of candidates that passed all tests. Two decades earlier, Massalin's superoptimizer [Massalin 1987] only used testing to reject machine code sequences; at the time the work was done, automated decision procedures were not advanced enough to finish the job of verifying candidates, and so this job fell to the superoptimizer's users.

Outside of superoptimization, the strategies used for guiding program synthesis and pruning the search space show rich domain specific variation. Tiwari et al. [2015] describe a dual interpretation scheme for constraining the search space for an exists-forall problem to synthesize encryption schemes. In this case, not only does the synthesized program have to equivalent to the specification, it also has to conform to certain security properties. The SuSLik [Polikarpova and Sergey 2019] paper describes the synthesis of heap-manipulating program. Here, synthesis is presented as proof search in Synthetic Separation Logic. The search space is pruned by ruling out unsolvable proof goals. The Unagi team describes prioritizing unbalanced trees for an ICFP programming contest [Akiba et al. 2013].

7 CONCLUSION

By speeding up an enumerative synthesis strategy, we have made Souper, an open-source superoptimizer for LLVM IR, about 2.3x faster when synthesizing peephole optimizations for around 269,000 specifications derived from compiling all of the C and C++ applications in the SPEC CPU 2017 benchmark suite, while also supporting an 8% increase in synthesis successes, where a candidate that has lower cost than the specification is found within a 300 second timeout. The key technique that we developed is a collection of dataflow analyses that extract properties of the synthesis specification and of partially symbolic candidates—which contain symbolic constants and also uninstantiated instructions—in order to establish a conflict between the specification and a candidate. When a conflict exists, the candidate can be pruned without resorting to calling an SMT solver. Speeding up synthesis is important because synthesis is very expensive; making it faster allows a tool like Souper to look further into the search space, to find useful optimizations.

ACKNOWLEDGMENTS

This material is based upon work supported by the Office of Naval Research under Grant No. N00014-17-1-2996.

REFERENCES

Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michał Moskal, and Nikhil Swamy. 2013. Calibrating Research in Program Synthesis Using 72,000 Hours of Programmer Time.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. *Formal Methods in Computer-Aided Design*, 1–17. https://doi.org/10.1109/FMCAD.2013.6679385

Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA). 394–403. https://doi.org/10.1145/1168918.1168906

Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. 2019. Extending enumerative function synthesis via SMT-driven classification. 212–220. https://doi.org/10.23919/FMCAD.2019.8894267

Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. 123—-134. https://doi.org/10.1145/2442516.2442529

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. 775–788. https://doi.org/10.1145/2837614.2837666

Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani. 2006. To Ackermann-Ize or Not to Ackermann-Ize? On Efficiently Handling Uninterpreted Function Symbols in SMT. In *Proc. of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Phnom Penh, Cambodia). 557–571. https://doi.org/10.1007/11916277_38

Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. 653—-663. https://doi.org/10.1145/2568225.2568250

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. 62–73. https://doi.org/10.1145/1993316.1993506

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (Dec. 2019). https://doi.org/10.1145/3371080

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3062341.3062343

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 436–449. https://doi.org/10.1145/3192366.3192410

Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. 2014. UpSet: Visualization of Intersecting Sets. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 1983—-1992.

Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. 958–968. https://doi.org/10.1145/3180155.3180211

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS '87)*. 122–126. https://doi.org/10.1145/36177.36194

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. 297–310. https://doi.org/10.1145/2872362.2872387

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (Jan. 2019). https://doi.org/10.1145/3290385

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL]

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. 305–316. https://doi.org/10.1145/2490301.2451150

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of Machine Code from Semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. 596–607. https://doi.org/10.1145/2737924.2737960

Venkatesh Srinivasan, Tushar Sharma, and Thomas Reps. 2016. Speeding Up Machine-code Synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. 165–180. https://doi.org/10.1145/2983990.2984006

Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. 2015. Program Synthesis Using Dual Interpretation. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 482–497.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. 287–296. https://doi.org/10.1145/2491956.2462174

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. 283–294. https://doi.org/10.1145/1993316.1993532